(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
10 May 2001 (10.05.2001)

PCT

(10) International Publication Number
WO 01/33360 A1

(51) International Patent Classification[7]: G06F 11/36

(21) International Application Number: PCT/US00/29122

(22) International Filing Date: 20 October 2000 (20.10.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/433,897 4 November 1999 (04.11.1999) US

(71) Applicant: TERADYNE, INC. [US/US]; 321 Harrison Avenue, Boston, MA 02118 (US).

(72) Inventor: GLIK, Michael, V.; 44 Sharpe Road, Newton, MA 02459 (US).

(74) Agent: WALSH, Edmund, J.; Teradyne, Inc., 321 Harrison Avenue, Boston, MA 02118 (US).

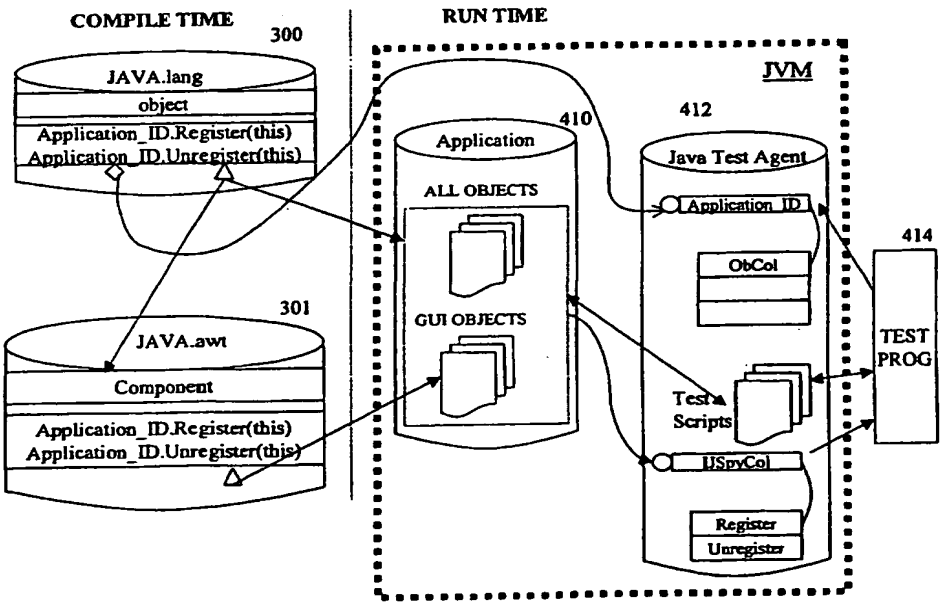
(81) Designated States *(national)*: AU, JP, SG.

(84) Designated States *(regional)*: European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published:
— *With international search report.*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: SIMPLE METHOD FOR TESTING PLATFORM INDEPENDENT SOFTWARE



(57) Abstract: A computer system that includes platform independent software that can be easily tested. Modifications are made to selected objects in the basic library associated with the platform independent language. These modifications allow the test software to monitor interactions between the applications software and the virtual machine implementing the platform independent language. In this way, a single baseline can be used for testing of any platform, thereby greatly simplifying the process of testing, particularly when many different platforms are used within an enterprise.

WO 01/33360 A1

SIMPLE METHOD FOR TESTING PLATFORM INDEPENDENT SOFTWARE

This invention relates generally to software systems and more particularly to testing software programs.

In recent years, software has become a major component of many products and systems. Testing software has therefore become more important. Several software test products are known. One such product is called ATF, distributed by Softbridge, Inc., Burlington, MA, USA. ATF is intended for testing graphical, multi-user, client/server applications, but software geared towards testing other applications is known.

ATF is a test program that includes a centralized program that controls testing across many computers in a network. Separate test code, called "agents," is loaded in the client computers in the network. During a test, the agents communicate with the centralized program to execute a suite of tests. In this way, the client and server computers throughout the network can be coordinated to simulate actual operating conditions in the network.

Each agent – like most software - must be written for the specific platform on which it operates. Thus, there are various types of agents, such as those for Windows® 3.*x*, NT, or 95. Other agents are available for OS/2® or WIN-OS/2™. The agent acts as the interface between application programs running on the client computers and the centralized test control program. The agent can do such things as determine the entries made by a human operator or determine what is being displayed on the screen of that computer. It is possible for the agent to ascertain this information because user interactions pass through the operating system of the client computer. Because the agent is operating system dependent, it knows how the operating system will represent this information. For example, the agent might read the memory buffer in which the operating system stores keyboard inputs from a user. Or the agent might read the memory structure in which the operating system stores the bit map of information on the screen for the user to read. The operating system stores various other data that provides information on the running program, which an agent might also access.

To perform a test on the applications running on the network, the agents read values from the specific client computers. This data can be compared to data values that were expected that the computers would be storing. The expected data values are sometimes called a "baseline." Differences from the baseline often indicate a fault in the system.

A limitation of this type of testing is that tests are generally only able to run based on the inputs and the outputs of the running application that are processed through the operating system. The test agent has no ability to determine the internal state of the applications program. For example, sometimes when a user enters data into a running application, the data is simply stored for later use. No immediate visible output is produced. More complete testing could be performed if the test agent could have access to the running application to determine its internal operating state. Heretofore, access to running applications was generally available only through customization of the application program or specialized knowledge about the application built into the test agent. There was no easy way to have access to every application program that might run on a network.

Another development in the software industry is called platform independent code. Platform independent code is code that is can be executed on computers that have different operating systems. For example, platform independent code might be executed on computers running the Windows® operating system or UNIX operating system. The operation under each operating system should be the same to the user.

The most prevalent example of platform independent code is that written in a programming language called JAVA®, distributed by Sun Microsystems of Palo Alto, CA. Java code is sometimes referred to in different ways, for example Java Applications, or Applets, Servlets, Java Beans, or Enterprise Java Beans. When a distinction is drawn in the name given to the code, it is often to signify either the size of the code or its intended use. Herein, all of these types of programs will be referred to simply as applications.

To prepare platform independent applications, an application developer requires a JAVA® compiler and a JAVA® basic library, which are referred to as RT.Jar or Classes.Zip, depending on the version of the language that is used. The basic library contains objects that perform various functions that are likely to be used in an application. For example, the basic library contains objects that define a dialog box that would display a message to a user of the application. There would be objects to create drop down lists that allow the user to pick a choice from the list. Many other objects are included in the basic library. One of the objects is called "java.lang.Object." This object is the root of all sub-classes for any object developed in the Java language.

The application developer then writes an application by creating application specific objects. The application specific objects can incorporate objects from the basic library.

When the application is completed, it is compiled using the compiler. The compiler creates an object that represents the application, including those objects from the basic library that are used by the application specific objects. In JAVA®, the compiled application is said to be in BYTE code.

The BYTE code is platform independent. To execute the BYTE code, it is provided to a computer that is configured as a JAVA® virtual machine. A JAVA® virtual machine is a computer application that is serving as an interpreter that manages all interaction with the operating system or hardware on that specific machine. In particular, it will translate the required interactions with memory, disks, communications ports and other hardware from the platform independent format of the BYTE code to commands the specific operating system understands.

A further testing difficulty arises with platform independent code. That code can be run on many different platforms. Substantial effort is required to be able to test the platform independent code on all the platforms on which it might be running. Separate agents are required for each platform – which must be created with some effort.

Because each platform might represent the inputs and outputs of the programs differently, it is sometimes necessary to have separate baseline values stored for each platform. The software used to run tests, sometimes called "test scripts," might also have to be different depending on which platform is to run the application. To fully test an application, numerous test scripts are required and baseline values are necessary for numerous test conditions. Thus, repeating this work of generating baseline values and test scripts for every platform that might be used to run the platform independent code would be very time consuming. Even on the same platform, the platform independent language might be implemented differently, further multiplying the test work required based on the number of vendors of the platform independent language.

We have discovered a simple way to allow the testing of platform independent applications.

SUMMARY OF THE INVENTION

With the foregoing background in mind, it is an object of the invention to provide a way to efficiently test platform independent applications.

It is also an object to allow scripts for testing applications written in a platform independent language to be constructed in a way that makes them independent of the platform on which the application runs.

It is also an object to allow tests of applications written in a platform independent language to be constructed in a way that makes them independent of the vendor who provided the platform independent language on the client computer.

The foregoing and other objects are achieved by providing a platform independent language with a basic library. The basic library has a plurality of program elements in it that is incorporated into the platform independent application. A portion of the program elements have incorporated therein code which provides access to a run time test program. During execution, the application, because of the code incorporated into it through the program elements in the library, provides information about its performance to the run time test program. The test program uses this information to detect faulty operation of the program.

In a preferred embodiment, access to the running applications is provided through an interface that is accessed from test scripts themselves written in the platform independent language.

In a preferred embodiment, the code embedded for test access is included in only a few of the program elements. The program elements selected for inclusion of the test access code are those which are necessarily included in any application.

In a preferred embodiment, the basic library is created by modification of the basic library for a commercially available platform independent language.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention will be better understood by reference to the following more detailed description and accompanying drawings in which

FIG. 1 shows a network employing the invention;

FIG. 2 is block diagram showing the software architecture of one of the client computers in the network of FIG. 1;

FIG. 3A and 3B illustrate modifications to platform independent language to facilitate test access;

FIG. 4 is a sketch illustrating the interaction of an application under test, a test agent and the basic library of a platform independent language.

DESCRIPTION OF THE PREFERRED EMBODIMENT

FIG. 1 shows a networked computer system 100 in which the invention might be employed. The system is made up of a network 110. Network 110 might be a local area network, or LAN, using Ethernet, 10baseT or other hardware configuration. Network 110 might alternatively be a wide area network. The invention is intended to operate regardless of the type of network that is used.

Various client computers, such as 112 and 114 are connected to network 110 and can communicate with each other to collectively implement an application. For simplicity, only two such client computers are shown. Also, client computers 112 and 114 are illustrated to be of different type. An important characteristic of the invention is that it can operate on networks with different types of computers.

Networked computer system 100 also includes test control computer 116. Test control computer 116 is connected to network 110, like the client computers 112 and 114. However, test control computer 116 is intended to execute tests. Specifically, it can receive data from test agents in each of the client computers 112 and 114. The received data can be compared to baselines developed for properly operating software, thereby detecting problems with the application.

Some or all of the client computers 112 and 114 execute applications written in a platform independent language. In this way, the application program can be written once and executed regardless of the specific type of computer connected to the network. It will be appreciated that test control computer 116 could also run a test application written in a platform independent language. In the preferred embodiment, the platform independent language is JAVA®.

FIG. 2 shows the architecture of any of the computers on the network using a platform independent language. At the lowest level, the architecture includes the specific computer hardware 210 in that computer. The computer hardware might be a personal computer from Compaq® or Dell® or other computer vendor. Alternatively, the computer hardware might be a work station from SUN Microsystems, AS-400 from IBM, HP-9000 from HP or other similar computer. Such hardware is well known in the computer industry.

The next layer of the architecture is the Bios and Operating system 212. This software interacts directly with the computer hardware. Bios and Operating system 212 might be the Windows® operating system or could be the UNIX operating system, VMS, Apple OS or other similar operating system. Such software is well known in the computer industry.

The hardware 210 and the Bios and Operating System 212 define the "platform" for the computer. The platform can be unique for each computer in the network. To run platform independent applications, there must be a common interface to the platform. The next layer in the architecture, interpreter 214, provides this common interface.

In the preferred embodiment, the platform independent software is JAVA®. Thus, the interpreter is called a JAVA virtual machine, or JVM.

The top layer of the architecture is application layer 218. Here, application layer 218 contains application 218a and 218b. Applications 218a and 218b are applications intended to be run on computer system 100. For example, they could be programs that allow a computer operator to enter data into a database. However, the functions performed by these applications is not important to the invention. Applications 218a and 218b contain user written code.

One of the ways that the user written code performs the desired functions is by invoking standardized functions that are part of the "language extension package" 216. Language extension package 216 is a collection of objects gathered from standardized libraries that are part of the platform independent language. These objects interact with the underlying computer hardware through interpreter 214. According to the invention, though, language extension package 216 has been modified to facilitate testing. The modifications are explained in greater detail below.

Also, application layer 218 contains test agent 218c. Test agent 218c tests the applications, such as 218a and 218b. To test these applications, test agent 218c must verify that the execution of applications 218a and 218b is causing the underlying hardware 210 to perform the correct operations at the appropriate time.

Test agent 218c tracks the interaction of the applications with the underlying hardware through the modified language extension package. The modifications made to facilitate this interaction for the preferred embodiment are shown in more detail in FIG. 3A and 3B. It should be appreciated that the specific modifications that are required will depend on the particular platform independent language and might even depend on the particular implementation of that language. For purposes of illustration, the preferred embodiment is described in conjunction with the JAVA language as implemented by Sun Microsystems.

FIG. 3A illustrates a portion of the language extension package, called package.java.lang. Within this package, a class Object is declared. Edits were made to this particular piece of the language extension package because Object is the root class in the language. Any object written using the language will inherit the methods

and properties of the class Object. Methods are program segments that implement specific functions. Properties can define the specific format to be used to store specific pieces of data or could be the variables that identify specific pieces of data that have been stored. When memory storage is actually allocated to store data, it is called an "instance" of the variable or it is said that the variable is "instantiated."

Application programs are objects. Thus, they inherit the methods and properties of the class Object. Thus, they will "know" about the methods and properties that are included for test access. Line 310 defines a property that is added for test access.

Line 310 indicates that a variable Application_ID should be set up. Further, when this variable is set up, it will initially be assigned a value of "null." This variable is indicated to be of type IJSpyCol, which is explained below. Suffice it to say here that this type is declared, also as a modification to the package.java.lang. Variables of this type, including Application_ID, store a description of a running application program. More specifically, it stores the information that a test agent would need to uniquely identify that application and also access the methods and properties of that application.

The value of the variable Application_ID is initially set to null. However, once the test agent becomes aware of the application or object in which this variable is instantiated, the test agent can change the value from "null" to a value which specifically identifies the application. However, in the preferred embodiment, the test agent changes the value from null only when the application or object is to be monitored for test purposes. In this way, the value null indicates either that the object has not yet been identified to the test agent or that that the object need not be tested. Separate variables could alternatively be used for these functions or other ways to signal these events could be used.

The next line of code that has been added for test access is line 312. This line is defined within a method that is performed when objects are terminated. Termination routines are part of almost all object oriented programming languages. In a Java Virtual Machine, the termination routine uses a "garbage collection" approach. Part of what the Java Virtual Machine does while running programs is keep track of which objects are being used by other objects. If one object is no longer being used by another object, the Java Virtual Machine knows that the object can be deleted. The Java Virtual Machine can then reuse the memory locations allocated to the object. However, before deleting the object from memory, the Java Virtual Machine executes any finalize method that has been specified for that object.

Line 312 defines the portion of the finalization method that will notify the test agent that the application program or object is no longer present. In this way, the test agent will stop testing that application or object. Line 312 contains an "if" statement. It is run only if the variable this.Application_ID has been assigned a value. As mentioned above, a value is assigned to this variable only if the test agent has determined that that the specific object needs to be monitored for testing. The prefix "this" is a JAVA keyword that refers to the object in which the method is actually used.

If the object is being monitored for testing, the method Unregister is invoked. This method has as an argument the keyword "this," which again identifies the specific object invoking the method. In this case, the object invoking the method is the object being terminated. The method Unregister notifies the test agent that the object is being terminated. The test agent will then stop monitoring that object and can make other checks as might be appropriate based on the specific test being run.

The next line that has been added is line 314. This line declares an interface to an object called IJSpyCol. As illustrated in FIG. 3A, this interface identifies two methods, Register and Unregister. These functions were mentioned above. When invoked, these methods either notify the test agent that the Java Virtual Machine is starting up or terminating an object. Line 314 does not actually invoke these methods; it simply states that such methods exist and can be found in IJSpyCol. The variable oMe simply represents the object that is calling the function, i.e. the object being tested.

IJSpyCol describes the interface to Register and Unregister as opposed to actually specifying those objects. The actual code that performs these functions will be provided as part of the Test Agent. The structure of the code in the test agent that is accessed with these interfaces will depend on such things as the way in which the test agent is implemented. However, as long as these functions have the specified interface, these functions can be used in conjunction with any application compiled using a modified basic library as shown in FIG. 3A. When compiled applications code is run in an environment with a test agent, it will correctly interface to the test agent. In this way, test access is provided to the application.

IJSpyCol is thus a class that represents the testing program to the interpreter for the platform independent language. In the present example, where the platform independent language is Java, it is representing the testing program to the Java Virtual Machine (JVM).

To facilitate testing, in the illustrated embodiment, modifications are also made to the object in the basic library called JAVA.awt.component. These modifications are shown in FIG. 3B. Line 350 indicates that a method JsunSpy.main1() has been added to the object Java.awt.component. As described above, the modifications to package.java.lang identify the functions Register and Unregister. They also cause the function Unregister to be invoked during a termination routine. They do not, however, invoke the function Register when an object is instantiated. In the illustrated embodiment where the JVM is implemented by Sun Microsystems, security restrictions cause the JVM to cease functioning correctly if the function register were invoked through modifications of package.java.lang. Therefore, JAVA.awt.Component has been modified to ensure that the method Register is invoked at the instantiation of each object. All GUI objects depend from JAVA.awt.Component. Thus, the modifications described in conjunction with FIG. 3B affect every object with a graphical user interface. We have observed that other implementations of the Java Virtual Machine do allow modifications to package.java.lang to cause the invocation of a method at object instantiation. Where execution will be limited to JVM that allow it, the changes described in conjunction with FIG. 3B could alternatively be incorporated in the package.java.lang.

Line 350 defines a static block. In the JAVA language, "static" means that the code segments within the block are executed once and only once during the initialization of the first object of this class. Thus, line 351 causes the function "main1" to be called during the initialization of the first object in class java.awt.Component or any of its subclasses. Thus, "main1" will be called before any of the objects to be tested runs.

The function "main1" is part of the object JSunSpy. This function call requires that JSunSpy be created. Thus, before any object to be can run, the object JSunSpy must be created. JSunSpy is an object that is provided as part of the test agent. The developer of the test agent specifies the events that happen during the construction of this object. Those events set up the test agent to conduct testing.

The specific steps that need to be followed will depend on the specific test agent. It is contemplated that existing test agent technology will be used to provide a test agent. However, the specifics of what the test agent is and does will depend in large measure on what type of tests are desired to be run.

Also during the initialization of JSunSpy, steps can be taken to customize the interface if necessary for either the platform independent language or any peculiarities

of the platform. For example, if it is determined that one vendor does not truly implement the platform independent language according to a defined standard, the initialization of the JSunSpy could include in the invocation of procedures that check which vendor provided the interpreter 214. Based on which vendor provided the interpreter, different functions might be called in setting up the test agent so that the test agent will generate commands that are appropriate for the particular interpreter. However, it is important to note that these customization steps are the result of code within JSunSpy, which is contained within the test agent. Other than adding line 350, no changes are required to the basic library or to the code being tested in order to provide test access.

Line 352 indicates a function call to GetConnectionNow() has been added to the constructor for Component. This function will therefore be invoked when any object that inherits properties from java.awt is constructed. The function GetConnectionNow() is described below. Simply, though, it identifies an object to the test program when invoked. As all GUI objects inherit properties from this object, this function will be called whenever such an object is instantiated. In this way, the test program gains access to all GUI objects. Line 354 is the declaration of the function GetConnectionNow(). That function contains an IF statement at line 356.

Line 356 actually causes the method Register to be invoked. As described above, when this method is called, the test agent is made aware of the object and can decide whether to monitor that object for testing. The invocation of the method Register is embedded in an "if" statement. The "if" statement ensures that the method Register is invoked only once for each object.

Turning to FIG. 4, the manner in which the few changes to the basic library identified in FIGs. 3A and 3B allow test access in the preferred embodiment is illustrated. In the preferred embodiment, JAVA is used as the platform independent language. As described above, an application written in Java is first compiled. During the compilation process, information for compiling each object is derived from the code written for that object and the code written for objects from which that object depends. The information is derived in a hierarchical fashion, starting first with the root node and working down to the specific object being compiled.

Because the class definition of Object in the package java.lang is the root, the compiler starts there. The compiler includes in the BYTE code commands that allocate space for the variable Application_ID and sets it to null. The compiler also includes in the termination routine portion of the compiled BYTE code that will

execute line 312 and Unregister the object when the garbage collection routine determines it is no longer being used.

The object JAVA.awt.component is also a basic object. It is used in compiling all objects and falls next in the hierarchy. This object invokes the method Register at line 354. Thus, the compiler will incorporate into the BYTE code a call to the Register function.

The BYTE code is transferred to the specific platform that will run the application. There, the Java Virtual Machine executes the code. However, before code is executed, the objects that make up that code are constructed. This construction is done in a hierarchical fashion. In particular, line 350 requires that the Object JSunSpy be constructed before any of the objects being tested can be constructed. In this way, the test interface and test agent will be initialized before any calls are made to it. For example, memory space to store identifications of the particular applications being tested is allocated before it is necessary for the test agent to store those identifications.

As the application begins to run, the JVM will reach the part of the BYTE code that specifies the method Register should be invoked. The JVM invokes that method. Invoking the method simply triggers execution of a function within test agent 412. Test agent 412 receives as an argument to that function the Java identification of the object being initiated. The test agent determines whether it will need access to that object and, if so, fills in the information in the Application_ID variable. This information includes such things as the name of the object as well as a unique identifier for it. The unique identifier gives the test agent the ability to track a particular application from instance to instance of that application. The specific algorithm by which a unique identified is assigned to each application is not critical to the invention. Various algorithms may be used to create a unique identifier for an object. Hash values, though commonly used for this purpose, are not preferred. Hash values result in a unique identifier, but the same value might not be assigned to the same object from run to run. Also, the hash value may depend on the implementation of the JVM. Therefore, in the preferred embodiment, the unique identifier was constructed from class and order number for the object in the Container/Component hierarchy or the inheritance of the object. But, in this case, it may be a different object required to be treated as such.

In FIG. 4, the test agent is shown to have a Java piece and a test program 414 outside of Java. The test program can be a traditional prior art test program or similar

program modified to take advantage of the additional access features made possible by the invention.

Once an application has "registered" with the test agent, the test agent can operate in the same way as prior art test agents. In particular, the test agent can access the operating system of the computer actually running the application. By monitoring the operating system, the test agent can determine keystroke and mouse movement inputs to the application under test. It can also monitor screen display outputs from the application.

But, the test agent can also perform functions not previously possible in the prior art. In particular, the test agent can access the methods and properties of each object within the application. By accessing the properties, the test agent can do such things as determine internal operating states of the application. In addition, because the test program is aware of the Java representation of the objects in the application as a result of the Register function, it can access the objects through a standard Java interface. Because access is through a standard interface, the methodology of accessing the application objects for test does not depend on the platform or implementation of the JVM on which that application runs.

By combining the ability to access methods and properties of the objects with the ability to monitor inputs to the application, the test agent can run much more powerful tests or can test more quickly by, for example, observing a keyboard input and verifying whether an expected change to the internal state of the application has occurred. Various other tests can also be run once access to the application is possible.

By accessing the methods of an application, the test agent can cause the application to perform specific functions at controlled times. The test agent can then observe whether the expected outputs have been obtained. The ability to control the execution of the application program makes possible a wide range of tests that were not previously possible. For example, in the past, testing could only be based on values passed to Windows OS controls. The invention allows testing based on properties in a Java object that are not provided to the Windows OS controls.

Access to the objects that make up the applications is likewise provided simply using the platform independent language. Generally, there are standard interfaces to objects. In the preferred embodiment using the Java language, the application interface is referred to as the application public interface or API. The API is a set of functions that can call any object and access its methods and properties. The test agent is programmed to used the API each time it needs to access an object.

There could be many applications or objects running on a computer at any specific time. The test agent builds a collection of objects – one for each object it is monitoring. Each object in the collection has the Application_ID format. In this way, monitoring and testing can be done for every application running on the computer.

The test agent needs some way to determine that an application is no longer active. In a computer running JAVA applications, the JVM performs the function of determining when an object is no longer needed and removing it from memory. When the JVM determines that an object is no longer being used, it executes termination routing that has been built into the BYTE code for that object. As mentioned above, each object will inherit the termination code that has been included in the definition of Object in the modified package java.lang. That termination code includes execution of the Unregister method. Thus, upon termination, the JVM will invoke the Unregister method.

The Unregister method informs the test agent that the object does not exist any longer. The test agent can then take whatever steps are necessary to complete the testing of that object. The specific steps taken will depend on the type of application and the type of tests desired for that object.

Test program 414 is the program that contains the test logic. This program controls execution of the functions of the prior art test software. However, it uses the interface of the invention to perform these functions. In addition, it includes software that does the Register and Unregister functions. The interfaces to these functions are provided through IJSpyCol as described above.

FIG. 4 illustrates that there are test scripts within the JVM. These test scripts perform the testing functions of the prior art. The test scripts will perform the functions that exercise the specific software in the application. When they are to run is dictated by the control logic of the test program. Also, the results are passed back to the test program 414 for processing as was done in the prior art. However, these test scripts can be written in the platform independent language because the interface of the invention allows the application objects to be accessed using standard Java commands. In contrast to the prior art, then, these test scripts do not have to be re-written for each platform on which the application runs. Nor do they have to be rewritten if the JVM on different platforms are different.

From the foregoing, it will be appreciated that numerous advantages can be obtained. A very small amount of code – which could be as small as a single line – is required to allow a test application development company to have access to any platform independent application to enable testing of the applications.

Another advantage is that a company wishing to test an enterprise wide application that runs platform independent code on many different platforms in the network needs to keep only a single base line of a test suite, but can nonetheless test their application on every platform they use.

Also, the above described system can test any type of object. Testing is not simply limited to objects Graphical User Interface (GUI) representations or limited to testing the inputs and outputs of the application that run through the operating system.

A further advantage is that much of the test code can be written in the form of test scripts in the platform independent language. In this way, test software can be quickly developed for many configurations.

Having described one embodiment, numerous alternative embodiments or variations might be made. For example, the foregoing embodiment has described that an interpreter is used to create a virtual machine at run time. It is not necessary that the interpreter interpret the commands as they are being executed. It is possible that the interpreter could interpret all the commands in the BYTE object at one time. In that case, what has been identified as an interpreter could also be thought of as a compiler.

Also, the invention is described in conjunction with an object oriented programming language. Traditional languages might be used. Instead of objects, such languages would have library with procedures or functions that would be compiled or linked into an application program.

In addition, a specific platform independent language was described. In the preferred embodiment, specific objects in that language were identified as requiring change. It is not necessary that those specific objects be changed as the benefits of the invention might be achieved by changing other objects. As one illustration, it was described that java.awt.component was changed to incorporate the Register method. We have found that this is preferable when the compiler is provided by SUN. We have found that JAVA compilers provided by other sources will allow methods in java.lang.object to be executed on construction of an object. Thus, all the code added to java.awt.component might be included in java.lang.object.

It is also not necessary that either of these specific objects be modified to include the required Register and Unregister instructions in the object being tested. The same instructions might be included in any object from which the object being tested would inherit properties.

As an additional variation, the invention was illustrated in the context of a hierarchical object oriented language in which methods and properties were inherited from other objects. The preferred embodiment was able to provide the required test

access through modification of the basic library. Test access might be provided through modifications of the compiler or the run time interpreter. For example, the compiler could be modified to insert into the start up and termination routines of every object functions similar to the Register and Unregister functions. Or, the run time interpreter could be modified to invoke such functions upon the construction or deconstruction of every object.

Further, the invention was illustrated in conjunction with a networked computer system. However, the invention is not limited to networked computers. Platform independent language can be run on stand-alone computers as well and the invention would be useful with such computers.

Moreover, the invention was described in the context of testing application programs. There are other instances where it would be desirable to simply observe the interactions between a computer program written in an application independent language and the underlying hardware. One example would be to test whether the interpreter were actually performing correctly.

Further, the preferred embodiment was described for testing software that is run on an enterprise wide application. No such limitation is required. The invention could be used with software developed for stand alone computers or any other application.

Also, in the preferred embodiment, the test program was written in a language other than Java. Java test scripts were used. It should be appreciated that the entire test program could be written in the platform independent language. Alternatively, the test scripts might be written predominately in a language other than the platform independent language – though they must have an interface to the objects running in the application independent language.

Moreover, the preferred embodiment was described in connection with testing to determine the functionality of applications programs. The test access techniques of the invention could be used to test in other ways, such as to verify the operation of the interpreter. The specific tests that would be run depend on the programming of the test agent, which one in skill in the art could prepare based on the specific things to be tested.

Therefore, the invention should be limited only by the spirit and scope of the appended claims.

What is claimed is

1. A method of testing platform independent application software of the type having a basic library, the method comprising:
   a) providing a basic library with at least a portion of the objects in the library having code embedded therein for causing communication with a run time test program;
   b) compiling the application code with the basic library to produce a platform independent executable object;
   c) providing the platform independent object to a hardware platform having a platform specific interpreter and a run time test program;
   d) executing the platform independent object with the interpreter, thereby causing communication with the run time test program;
   e) diagnosing the operation of the platform independent application software based on the communication with the run time test program.

2. The method of testing platform independent application software of claim 1 wherein the platform independent application software is written in the JAVA language.

3. The method of claim 2 wherein one of said objects in the basic library is java.lang.

4. The method of claim 2 wherein one of said objects in the basic library is java.awt.Component.

5. The method of testing platform independent application software of claim 1 wherein the step of providing a basic library comprises providing a basic library with at least a portion of the objects in the library having code embedded therein for causing communication with a run time test program, wherein the embedded code is provided in a portion of the objects in the library which is less than half the objects.

6. The method of claim 5 wherein the portion of the plurality of objects having code embedded therein for causing communication with a run time test program consists of objects that are necessarily included in all platform independent objects.

7. The method of claim 1 wherein the step of diagnosing the operation of the platform independent application software comprises execution of test scripts written in the platform independent language.

8. The method of claim 1 wherein the step of causing communication with the run time test program includes the step of calling a procedure defined within the run time test program that records an identifier for the application software.

9. The method of claim 8 wherein the step of diagnosing the operation of the platform independent application software includes accessing internal properties of the application software using an interface defined by the platform independent language.

10. A software system encoded in computer readable media comprising:
a) a basic library, the library having a plurality of objects with at least a portion of the plurality of objects containing code to interface with a run time test program;
b) a compiler that produces platform independent code from application code, the platform independent code including objects from the basic library;
c) an interpreter adapted to translate the platform independent code into platform specific code.

11. The software system of claim 10 additionally comprising a run time test program that include procedures that run when invoked through the interface in the basic library.

12. The software system of claim 11 wherein one of the procedures of the run time test program computes a unique identifier for the application object that calls that procedure.

13. The software test system of claim 11 wherein on of the procedures of the run time test program stores an identifier for the application object that calls that procedure and the run time test program further includes software that uses the

stored identifiers to make calls through the interpreter to access internal variables of the application objects.

14. The software system of claim 10 additionally comprising a run time test program that includes a plurality of test scripts written in a platform independent language.

15. In a computer system of the type having an operating system, a test agent and an application object, a method of provide test access to the application program comprising:

a) during initiation of the application object, performing a function of identifying the application object to the test agent, including storing an identification of the application object;

b) monitoring, by the test agent, parameters of the operating system to determine input and outputs to the application object; and

c) accessing, by the test agent, internal operations of the application object.

16. The method of claim 15 wherein the step of accessing includes reading the value of data stored within the application object.

17. The method of claim 15 wherein the step of accessing includes invoking a method defined within the application object.

18. The method of claim 15 wherein identifying the application object to the test agent includes providing a library object from which the application object depends, the library object having therein registration code that executes when an application object depending from the library object is initialized, the registration code calling a procedure within the test agent.

19. The method of claim 15 wherein accessing the internal operations of the application objects is done using the stored identification of the application object and an application public interface for the computer language in which the application program is written.

20. The method of claim 15 wherein the application objects are written in JAVA and accessing the internal operations of the application objects is done using the stored identification of the application object and the JAVA Application Public Interface.

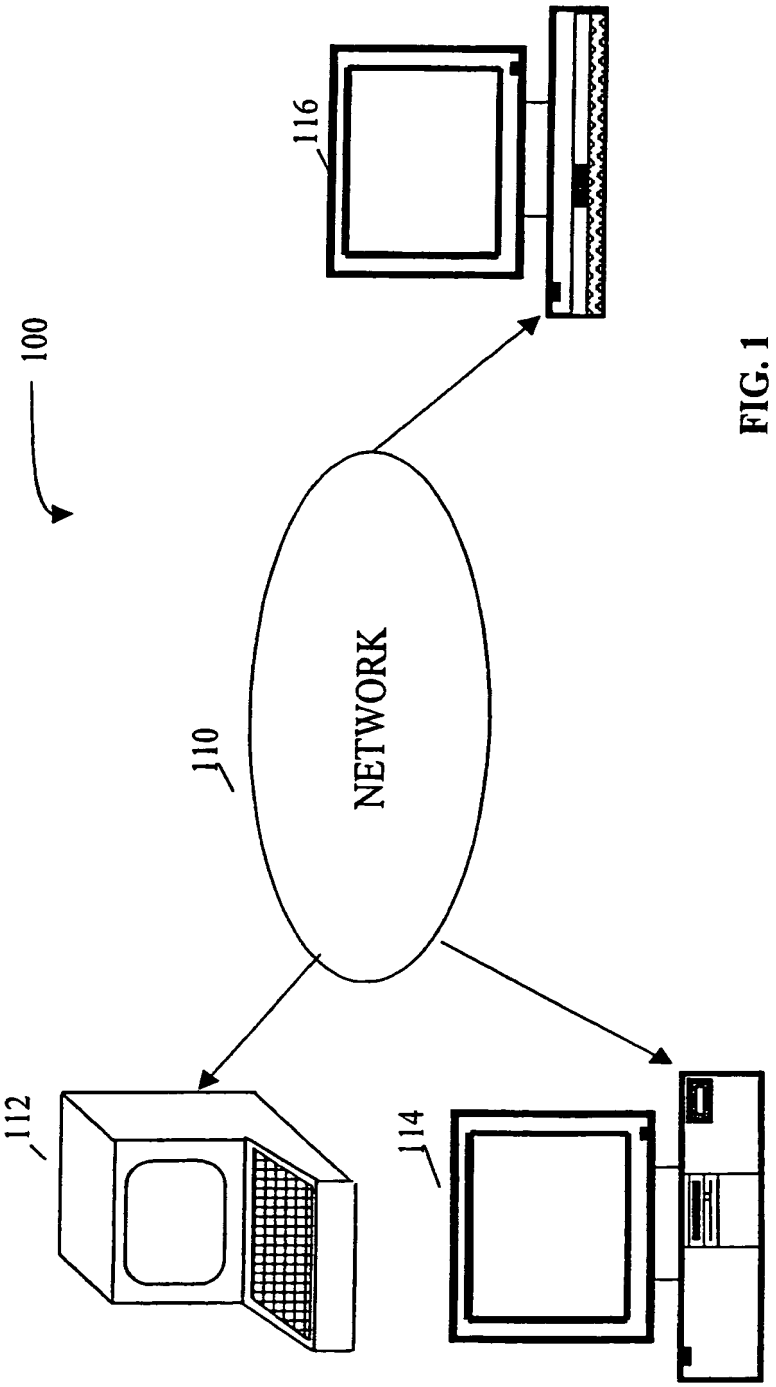


FIG. 1

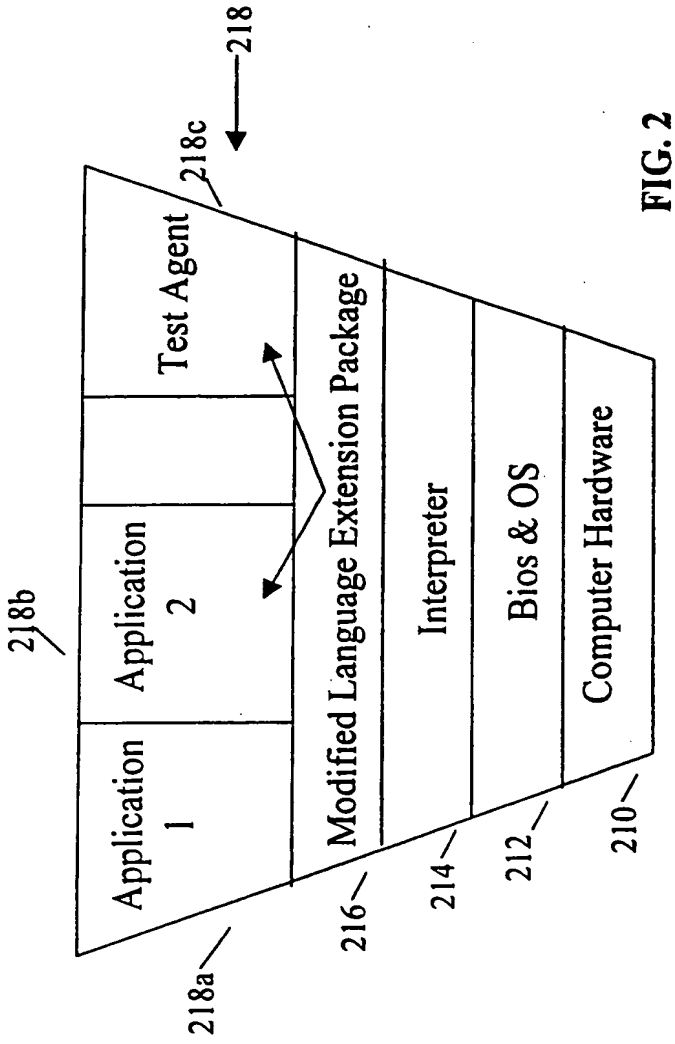

218
218c
Test Agent
218b
Application 1
Application 2
Modified Language Extension Package
Interpreter
Bios & OS
Computer Hardware
218a
216
214
212
210

FIG. 2

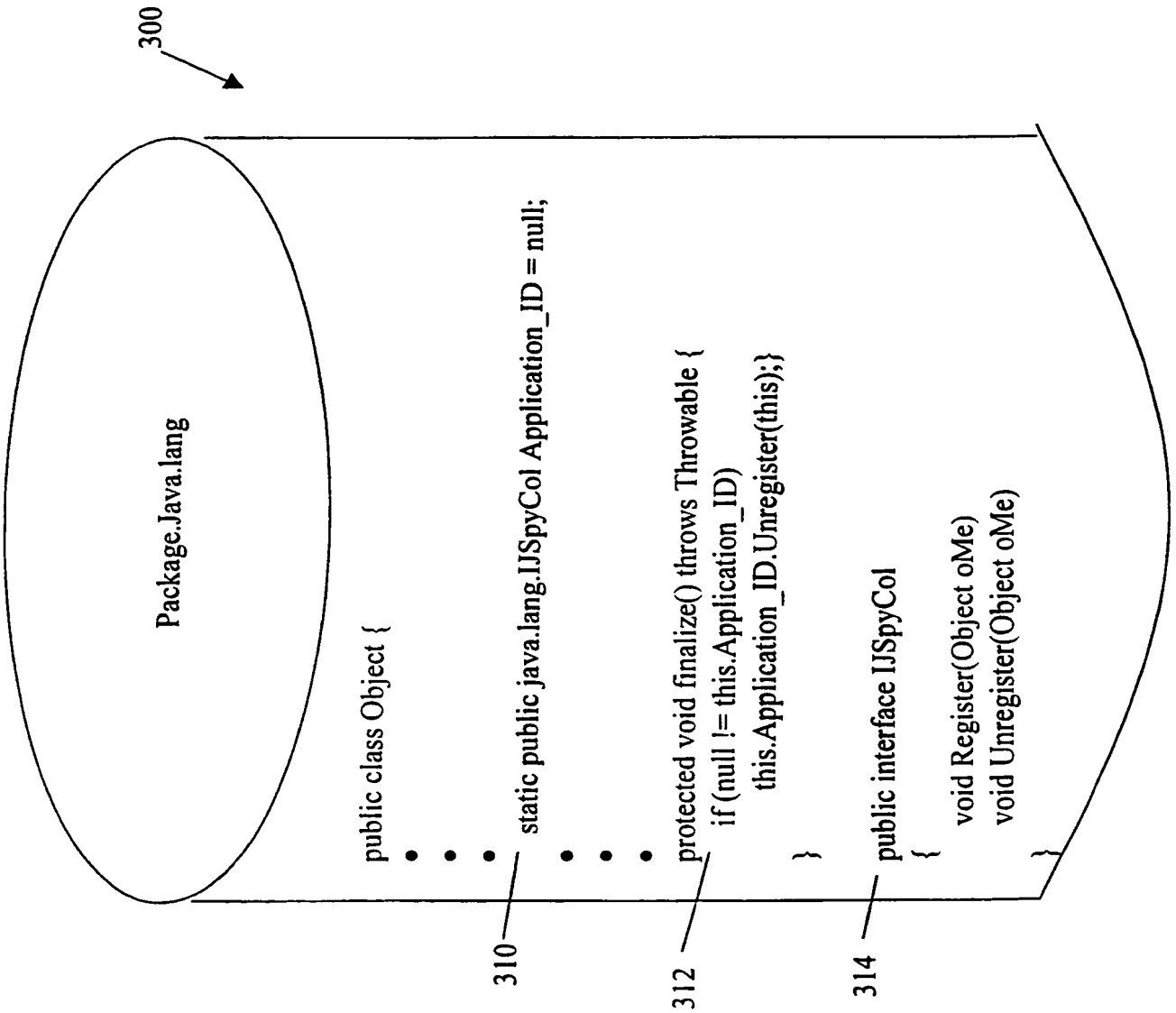

300
Package.Java.lang
public class Object {
static public java.lang.IJSpyCol Application_ID = null;
protected void finalize() throws Throwable {
if (null != this.Application_ID)
this.Application_ID.Unregister(this);}
}
public interface IJSpyCol
{
void Register(Object oMe)
void Unregister(Object oMe)
}
310
312
314

FIG. 3A

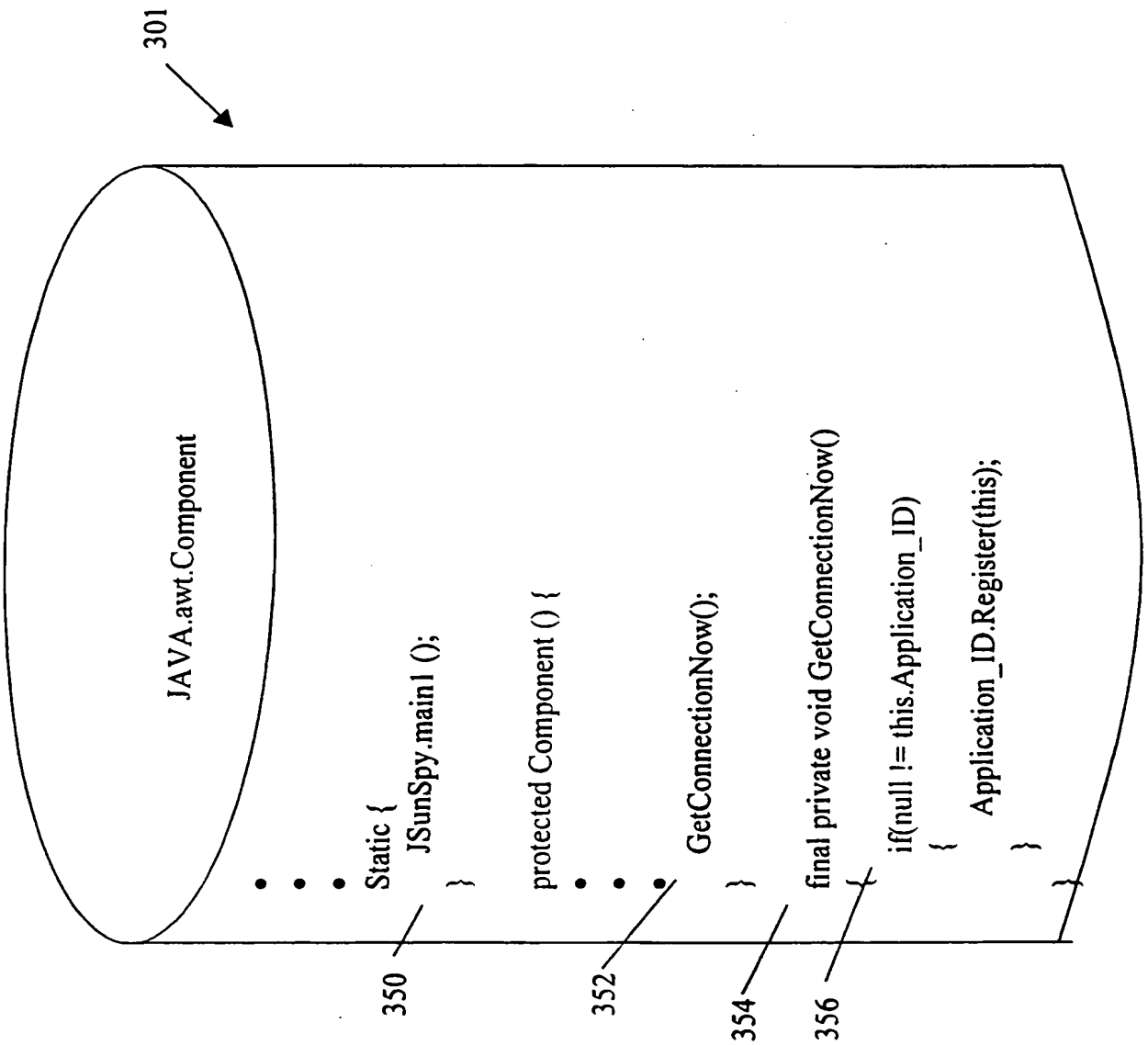

301
JAVA.awt.Component
Static {
JSunSpy.main1 ();
}
protected Component () {
GetConnectionNow();
}
final private void GetConnectionNow()
if(null != this.Application_ID)
{
Application_ID.Register(this);
}
350
352
354
356

FIG. 3B

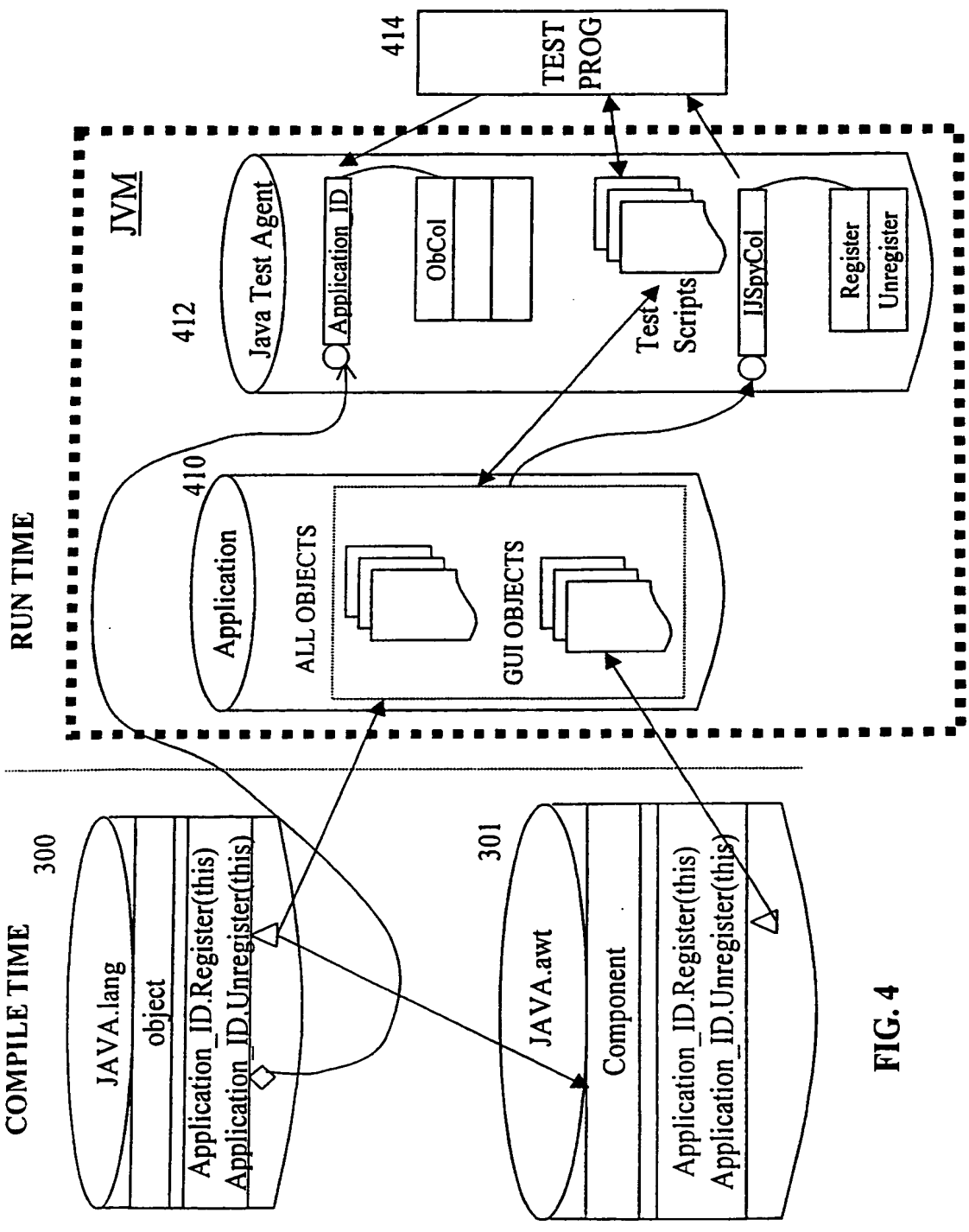

414
TEST PROG
JVM
412
Java Test Agent
Application_ID
ObCol
Test Scripts
IJSpyCol
Register
Unregister
RUN TIME
410
Application
ALL OBJECTS
GUI OBJECTS
COMPILE TIME
300
JAVA.lang
object
Application_ID.Register(this)
Application_ID.Unregister(this)
301
JAVA.awt
Component
Application_ID.Register(this)
Application_ID.Unregister(this)

FIG. 4

INTERNATIONAL SEARCH REPORT

Inter ›nal Application No
PCT/US 00/29122

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F11/36

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)
EPO-Internal, INSPEC, IBM-TDB, COMPENDEX, WPI Data, PAJ

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 5 781 720 A (KEPPLE LAURENCE RALPH ET AL) 14 July 1998 (1998-07-14)	15
Y	column 3,. line 32 -column 4, line 29 column 5, line 24 -column 8, line 36 column 11, line 38 -column 12, line 65 column 13, line 41 -column 17, line 53 --- -/--	1,2, 7-13,19, 20

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
12 January 2001	05/02/2001

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Renault, S

2

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Inte: onal Application No
PCT/US 00/29122

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
Y	GAISSERT, D.L. : "A Java based test program set development environment" AUTOTESTCON '99. IEEE SYSTEMS READINESS TECHNOLOGY CONFERENCE, 1999., 30 August 1999 (1999-08-30) - 2 September 1999 (1999-09-02), pages 775-785, XP002157102 San Antonio, TX, USA abstract page 779, column 2, line 7 -page 780, column 1, line 20 figures 2,3 ---	1,2, 7-13,19, 20
A	TYLER, D.F. : "Java-based automated test systems: management considerations for an open architecture for test " AUTOTESTCON '99. IEEE SYSTEMS READINESS TECHNOLOGY CONFERENCE, 1999., 30 August 1999 (1999-08-30) - 2 September 1999 (1999-09-02), pages 699-706, XP002157103 San Antonio, TX, USA abstract page 701, column 1 -page 703, column 2 ---	1-20
A	ORTON K L ET AL: "DESIGN AND DEVELOPMENT OF THE CDE 1.0 TEST SUITE" HEWLETT-PACKARD JOURNAL,US,HEWLETT-PACKARD CO. PALO ALTO, vol. 47, no. 2, 1 April 1996 (1996-04-01), pages 54-61, XP000591790 the whole document -----	1-20

2

INTERNATIONAL SEARCH REPORT

information on patent family members

Inter ›nal Application No
PCT/US 00/29122

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 5781720 A	14-07-1998	US 5600789 A	04-02-1997
		AU 5675094 A	08-06-1994
		WO 9411818 A	26-05-1994